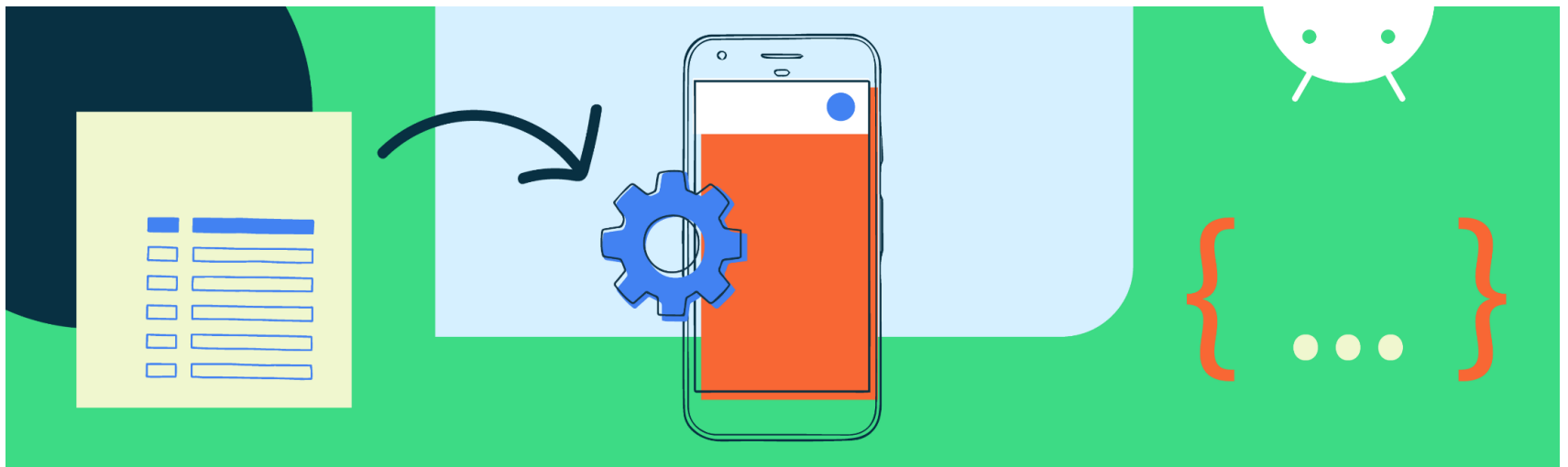
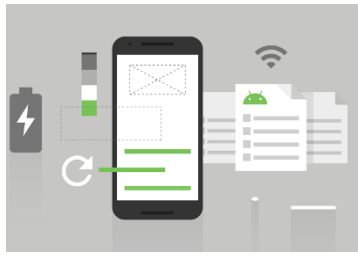


# Background processing using WorkManager





# WorkManager

- WorkManager is an Android library to **schedule & execute deferrable** background work
  - Deferrable = ok to run at later time
  - Intended for tasks that require a **guarantee** that the system will run them even if the app is inactive
- Can specify **constraints** that must be satisfied before the work is executed (e.g., only upload images to Firebase Storage when Wi-Fi connection is available)
- Can configure **retries** if the job fails

# Implementing Work Manager

- Add Dependency

```
def work_version = "2.7.0"  
implementation "androidx.work:work-runtime-ktx:$work_version"
```

- Extend **Worker** class
- Override **doWork** method
  - Return result: SUCCESS, FAILURE, RETRY
- Schedule Work: immediate execution, execute after initial delay, execute periodically

# Define work to do using Worker

- Define a unit of work to perform in the background using class that extends **Worker** class and implements **doWork** method

```
class UploadWorker(context: Context, params: WorkerParameters) : Worker(context, params) {
    override fun doWork(): Result {
        return try {
            val count = inputData.getInt(Constants.COUNT_VALUE , 0)
            for (i in 0 until count) {
                Log.i("UploadWorker", "Uploading $i")
            }
            val dateFormat = SimpleDateFormat("dd/M/yyyy hh:mm:ss aa")
            val currentDate = dateFormat.format(Date())

            val outputData = workDataOf(Constants.CURRENT_DATE to currentDate)
            Result.success(outputData)
        } catch (e: Exception) {
            Result.failure()
        }
    }
}
```

# One Time Work Request

- Create a **OneTimeWorkRequest**, pass parameters. Then **enqueue** the request
- Can start immediately or after an **Initial Delay**
- **.addTag** is used to assign a Human Readable identifier or create logical groups of work requests

```
val inputData = workDataOf(Constants.COUNT_VALUE to 125)
val downloadRequest = OneTimeWorkRequestBuilder<DownloadWorker>()
    .setInitialDelay(10, TimeUnit.Minutes)
    .setInputData(inputData)
    .addTag(Constants.TAG_DOWNLOAD)
    .build()
```

```
WorkManager.getInstance(applicationContext)
    .enqueue(downloadRequest)
```

# Schedule Period Work Request

- Use **PeriodicWorkRequest** to schedule a work to repeat periodically

*// Create a periodic work request with 15 mins as repeat interval*

```
val repeatInterval = 15
```

```
val periodicWorkRequest = PeriodicWorkRequestBuilder<DownloadWorker>  
    (repeatInterval, TimeUnit.MINUTES).build()
```

```
WorkManager.getInstance(applicationContext).enqueue(periodicWorkRequest)
```

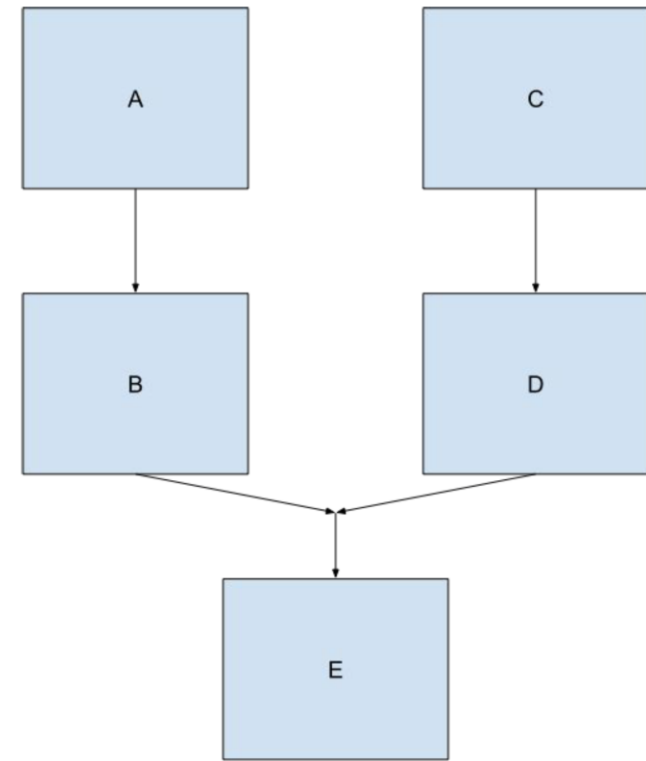
# Define Constraints

- You can define constraints that must be met before the work starts:
  - Network connectivity
  - Battery
  - Storage
  - Device State: device charging, device idle

```
val constraints = Constraints.Builder()  
    .setRequiredNetworkType(NetworkType.CONNECTED)  
    .setRequiresBatteryNotLow(true)  
    .setRequiresCharging(true)  
    .setRequiresDeviceIdle(false)  
    .setRequiresStorageNotLow(false)  
    .build()  
  
val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()  
    .setConstraints(constraints)  
    .build()
```

# Work Chaining

- Orchestration of multiple jobs. E.g.,
  - B runs after A
  - D runs after C
  - E runs after B and D are completed



```
val parallelWorks = listOf(downloadRequest, filterRequest)
workManager.beginWith(parallelWorks)
    .then(compressRequest)
    .then(uploadRequest)
    .enqueue()
```



# Configure retries

- If you require that WorkManager retry failed work, you can return **Result.retry()** from your worker. Your work is then **rescheduled** according to a **backoff delay** and **backoff policy**.

```
val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()  
    .setBackoffCriteria(  
        BackoffPolicy.LINEAR,  
        10,  
        TimeUnit.MINUTES)  
    .build()
```

# Unique Work

- Three possible policies for **OneTimeWorker**: KEEP, REPLACE, APPEND
- Two possible policies for **PeriodicWorker**: KEEP, REPLACE

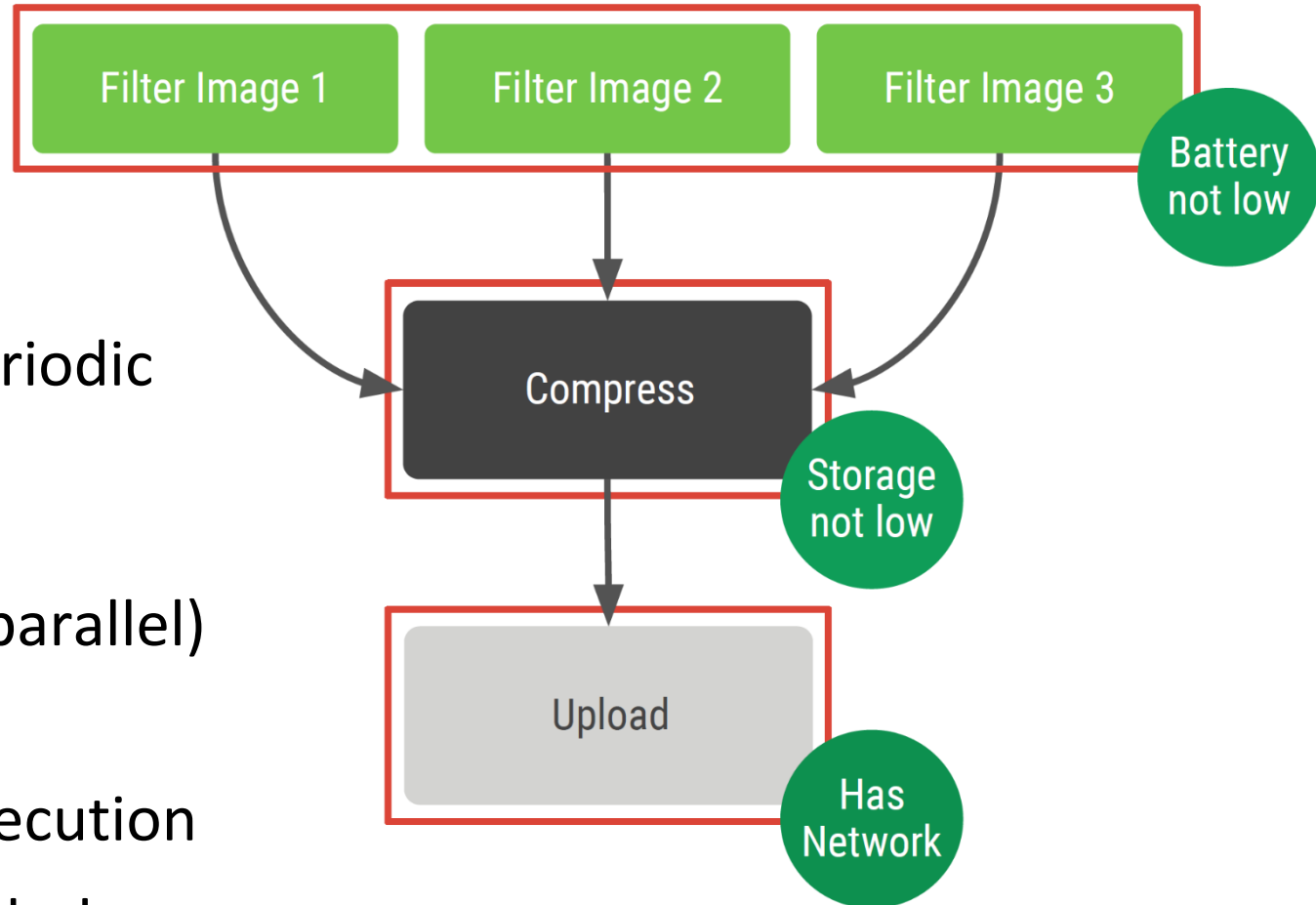
```
class MyApp: Application() {  
    override fun onCreate() {  
        super.onCreate()  
        val backupWorkRequest =  
            PeriodicWorkRequestBuilder<BackupWorker>(8, TimeUnit.HOURS).build()  
        WorkManager.getInstance(applicationContext).enqueueUniquePeriodicWork(  
            "BackupWork",  
            ExistingPeriodicWorkPolicy.REPLACE,  
            backupWorkRequest)  
    }  
}
```

# Coroutines + WorkManager

- Use **CoroutineWorker** to call coroutines in **doWork**
- You can specify a Dispatcher to use otherwise **Dispatchers.Default** is used by default

```
class AsyncWorker(context : Context, params: WorkerParameters)
    : CoroutineWorker(context, params) {
    override suspend fun doWork(): Result = withContext(Dispatchers.IO) {
        try {
            // Do async tasks
            Result.success()
        } catch (error: Throwable) {
            Result.failure()
        }
    }
}
```

# Summary of features



- One-off and periodic work
- Work Chaining (sequential or parallel)
- Constraints
- Guaranteed execution
- Query work state to display on UI

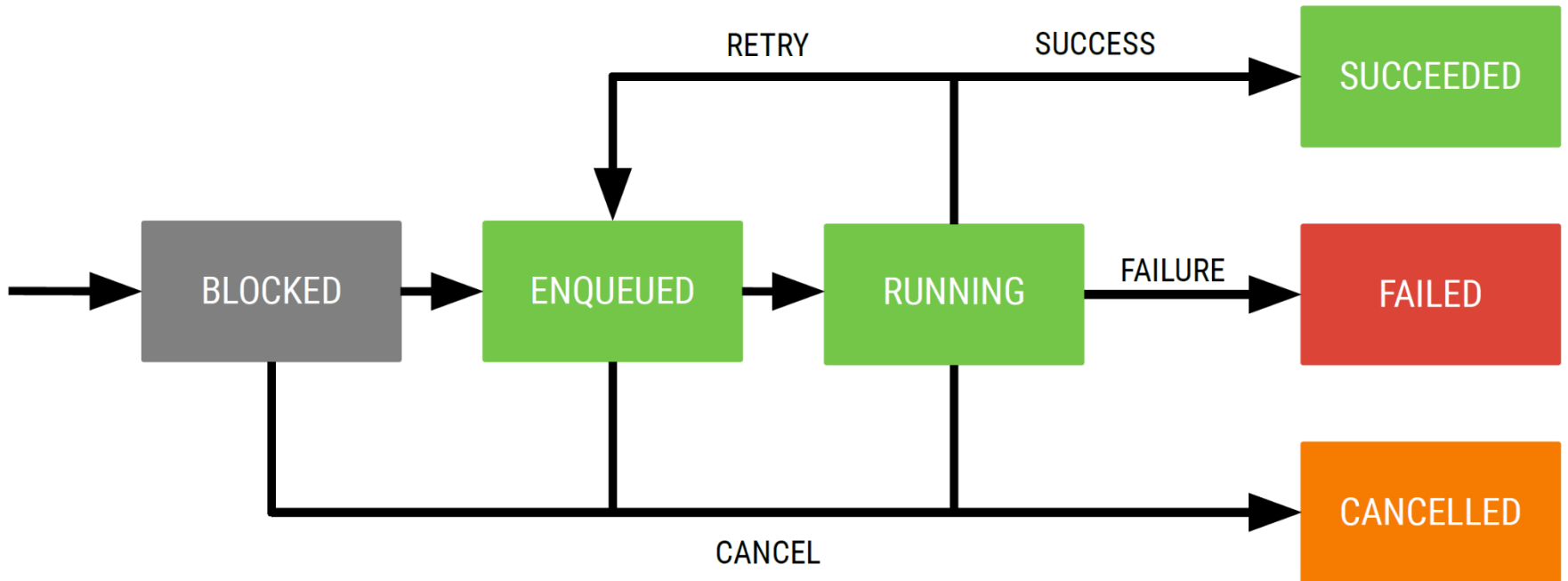
# Monitor work execution

- Query status by ID, Tag or Unique Name

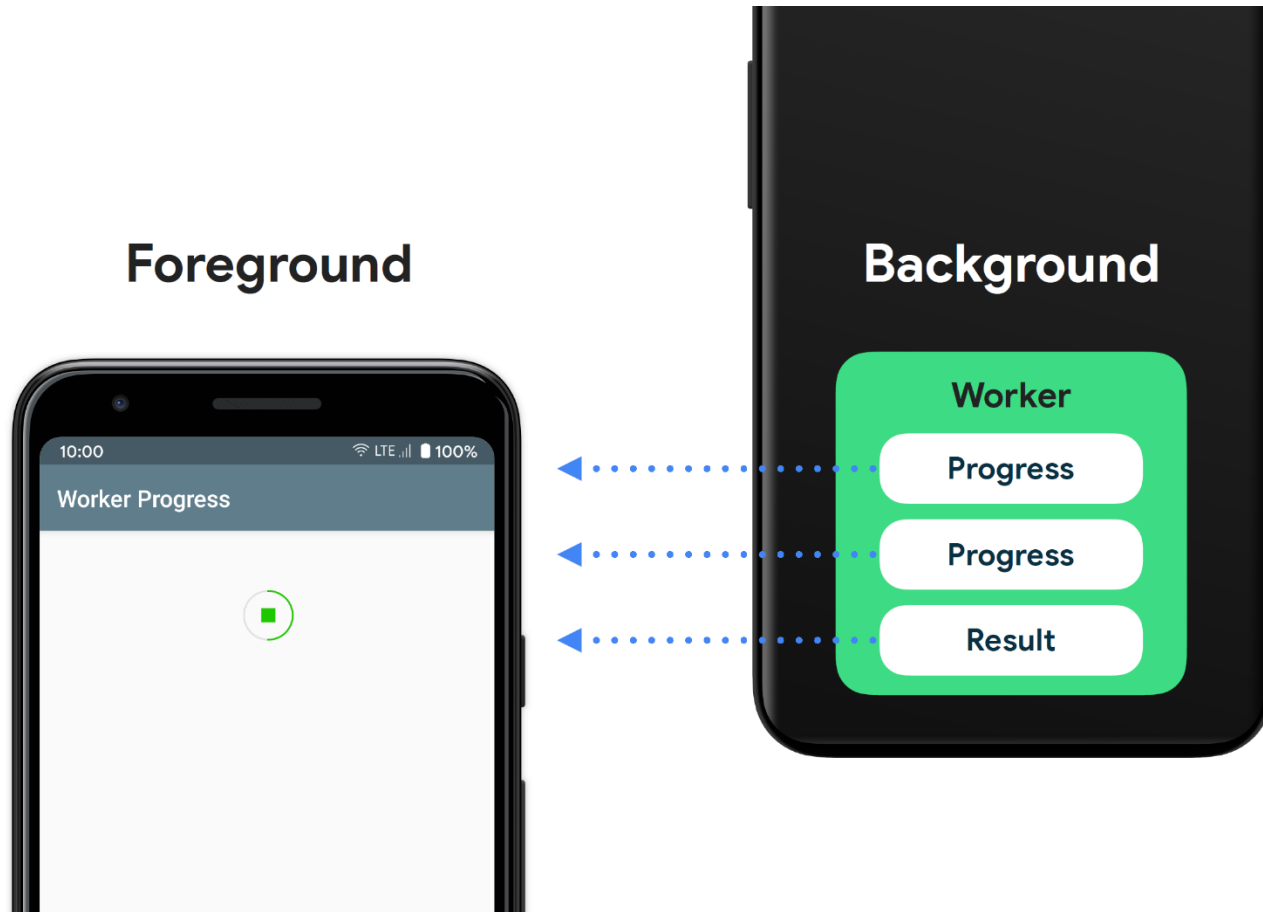
```
workManager.getWorkInfoById(requestId)
workManager.getWorkInfosByTag("Sync")
```
- Monitor status → LiveData providing job status
  - Use **.getWorkInfoByIdLiveData** to observe the work progress

```
workManager.getWorkInfoByIdLiveData(uploadRequest.id)
    .observe(this, Observer {
        textView.text = it.state.name
        if(it.state.isFinished){
            val data = it.outputData
            val message = data.getString(Constants.CURRENT_DATE)
            Toast.makeText(applicationContext, message, Toast.LENGTH_LONG).show()
        }
    })
```

# Life of OneTime Work



# Worker Progress



# Reporting Worker Progress

```
class ProgressWorker(context: Context, parameters: WorkerParameters) :  
    CoroutineWorker(context, parameters) {  
    override suspend fun doWork(): Result {  
        setProgress(workDataOf(Constants.PROGRESS to 25))  
        ...  
        setProgress(workDataOf(Constants.PROGRESS to 50))  
        ...  
        return Result.success()  
    }  
}
```



# Observing Worker Progress

```
val request = OneTimeWorkRequestBuilder<ProgressWorker>().build()
workManager.
    .getWorkInfoByIdLiveData(request.id)
    .observe(this, Observer { workInfo: WorkInfo? ->
        if (workInfo != null) {
            val progress = workInfo.progress
            val value = progress.getInt(Constants.PROGRESS, 0)
            // Do something with progress information
        }
    })
```

# Cancel Work

- Can cancel work using the work request id or the associated tag

```
val saveImageWorkRequest = OneTimeWorkRequestBuilder<SaveImageWorker>()  
    .addTag(TAG_SAVE_IMAGE)  
    .build()
```

```
WorkManager.getInstance(applicationContext).cancelWorkById(saveImageWorkRequest.id)
```

```
WorkManager.getInstance(applicationContext).cancelAllWorkByTag(TAG_SAVE_IMAGE)
```

```
// Or cancel all work
```

```
WorkManager.getInstance(applicationContext).cancelAllWork()
```

# Summary

- Schedule & execute deferrable background work
- Guarantees execution across system reboots
- Could be one-time or periodic work
- Cancellable work
- Can query the work state and progress

# Resources

- Getting started with WorkManager
  - <https://developer.android.com/topic/libraries/architecture/workmanager/basics>
  - <https://developer.android.com/topic/libraries/architecture/workmanager>
- WorkManager codelab
  - <https://developer.android.com/codelabs/android-workmanager>